BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYZER FOR LIST MANIPULATING PROGRAMS

NÁSTROJ PRO STATICKOU ANALÝZU PROGRAMŮ SE SEZNAMY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MICHAL KOTOUN**
AUTOR PRÁCE

SUPERVISOR **Prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Kotoun Michal**
Obor: Informační technologie
Téma: **Nástroj pro statickou analýzu programů se seznamy**
Static Analyzer for List Manipulating Programs
Kategorie: Formální verifikace

Pokyny:

1. Seznamte se s principy statické analýzy programů se seznamy s využitím symbolických paměťových grafů a implementací této analýzy v nástroji Predator.
2. Navrhněte architekturu nového prostředí pro statickou analýzu, které by mělo podporovat symbolické paměťové grafy, ale měly by do něj jít také doplnit další abstraktní domény.
3. Implementujte základ navrženého prostředí demonstrující jeho vlastnosti.
4. Otestujte vytvořenou implementaci na vhodných případových studiích.
5. Shrňte dosažené výsledky a diskutujte jejich další možný rozvoj.

Literatura:

- Dudka, K., Peringer, P., Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation, In: Proc. of SAS'13, LNCS 7935, Springer, 2013.
- The LLVM Compiler Infrastructure, online dokumentace, http://llvm.org/docs/.
- Dudka, K., Peringer, P., Vojnar, T.: An Easy to Use Infrastructure for Building Static Analysis Tools, In: Proc. of EUROCAST'11, LNCS 6927, Springer, 2011.
- Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, Springer-Verlag, 2005.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Vojnar Tomáš, prof. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

Creating a software verification tool is a complex task — one must implement source code parsing, instruction representation, value abstraction, user interface, ... and the analysis itself. Therefore, we decided to create a static analysis framework to prevent unnecessary wheel reinventing by an analyses implementers .

We propose a general design of the framework called Angie with a primary focus on usability, and describe a prototype implementation of the framework, including a model analysis based on symbolic memory graphs. Angie is implemented in C++ and uses the LLVM toolchain as the front-end for parsing the source code of analysed programs.

Abstrakt

Tvorba softwarového analyzátoru je komplexní úloha — je nutno implementovat parsování zdrojového kódu, reprezentaci instrukcí, abstrakci hodnot, uživatelské rozhraní, ... a také analýzu samu. Abychom předešli zbytečné práci vývojářů analýz, rozhodli jsme se vytvořit framework pro statickou analýzu programů.

Předkládáme obecný návrh frameworku zvaného Angie s důrazem na jeho použitelnost a popisujeme prototyp frameworku, včetně modelové analýzy založené na symbolických paměťových grafech. Angie je implementován v C++ a používá nástroje z kolekce LLVM pro parsování zdrojového kódu analyzovaných programů.

Keywords

static analysis, abstract interpretation, formal verification, LLVM, SSA, Predator, SMG, symbolic memory graphs, framework, Angie

Klíčová slova

statická analýza, abstraktní interpretace, formální verifikace, LLVM, SSA, Predator, SMG, symbolické paměťové grafy, framework, Angie

Reference

KOTOUN, Michal. *Static Analyzer for List Manipulating Programs.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

Static Analyzer for List Manipulating Programs

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Tomáš Vojnar. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Michal Kotoun
July 28, 2017

Acknowledgements

I would like to thank my fellow VeriFIT members, namely: my supervisor Tomáš Vojnar for his support both during work on the Angie project and in the critical time of writing this thesis; Angie project co-supervisor Petr Peringer for time during our lengthy dicussions; last but not least Martin Hruška and Tomáš Fiedor for moral support and impartial view when facing difficult design choices.

Contents

Chapter 1

Introduction

As the amount of software critically influencing our lives gradually increases, so rises the importance of different methods for checking the software for flaws. There are several well-known historical examples of what a software error can cause from the areas of space missions or pharmacy, and companies in such fields are still searching for the ultimate verification and bug-hunting tools. In this paper, out of the broad range of approaches that such tools can be based on, we are particularly interested in static analysis methods.

A static analyser for languages like C is a complex piece of software. It includes much more than just the analysis algorithms themselves: it must be able to parse all the details of the source language, provide a viable abstraction over the language data types and commands into some intermediate representation, and report results of the analysis in an acceptable format.

The aim of this thesis is to describe a new software analysis tool, or more precisely a framework for a family of such tools, inspired by a verifier for low-level C programs with dynamic linked data structures called *Predator*[1].

Predator is built as a GCC compiler plug-in, on top of the *Code Listener*[2] — an interface to access an intermediate representation of program from the *GNU GCC* (default) or *LLVM clang* [14] compilers. Its verification loop is based on abstract interpretation instantiated with *Symbolic Memory Graphs* [7]. Not only *Predator* is quite efficient and can handle many complex program constructions, it is also a multiple-time winner [12, 9, 2] of the *heap memory* and *memory safety* categories in SV-COMP[3].

The VeriFIT group — whose member the author is — would like to build on *Predator*'s success and push its usability border even further. However, this is not simple, considering its complex code-base.

Firstly, the architecture of *Predator* was designed to a large degree by a single developer who did not think much of later extensions of the tool and who left the team, and hence it is difficult to even understand all the details of the tool.

Secondly, *Predator* (and also the related tool *Forester* that shares with *Predator* the *Code Listener* infrastructure), is very optimized and any changes to it pose a great challenge (and use to pose it even to its original author).

Of course, one can start from the scratch — and do so every time when an entirely new analysis is needed — but doing so is far from optimal. If we consider what has already been said above about what all needs to be included into a static analysis tool, its is clear

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/

[2] http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/

[3] https://sv-comp.sosy-lab.org, International Competition on Software Verification

that the developer of the analysis will be forced to do a work similar to reinventing the wheel — by creating a waste number of base modules, like, e.g., front-ends, intermediate representation, abstraction domains etc.

Therefore, the conclusion has been that not only is a complete re-implementation of Predator needed, but that the group needs a platform for implementing static analyses — one that is easy for new developers to build upon. A framework.

The core requirements regarding the framework are listed below.

- Handles transforming instructions from a source language into some reasonable intermediate form.
- Provides analysis composition capabilities.
- Contains components for abstract interpretation support.
- Streamlines creating new analyses as much as possible.: it should not require a lengthy documentation study — or worse, an elaborate revere-engineering — prior starting the work on "your first analysis".
- Is under a good control of the group.

A work on such a framework, that we call *Angie*, is currently under way. We hope that it will make implementing a new program analysis for researchers appealing and simple. The initial scope for supported input languages should be C with possible extension for C++ later.

We designed Angie to be modular and not restrictive about the analyses that should be implemented in it — it is designed to allow combined analyses to be implemented in it. However, we do not intend to make it super-generic from the start, instead, we let it on the developers of the analyses to combine them.

We chose *LLVM*[4] to be the fronted since the *LLVM* tool-chain has a stable development and provides us with a way to support many input languages. The first and model analysis to be implemented in Angie is a shape analysis based on the *Symbolic Memory Graphs* originally introduced in Predator, to which we are making improvements allowing for a more precise abstraction.

We have chosen C++ to be the implementation language because it is one of the most used ones and because *LLVM* itself is implemented in it. We are aware of the fact, that C++ might not be the easiest language to use when implementing static program analysis, but we hope this choice will prove to be acceptable as students and young researchers with an existing programming background should be reasonably familiar with the concepts we use.

We have already successfully implemented an *LLVM* front-end adapter which wraps the ever-changing *LLVM API*, defined an interface for combining abstract value domains, and created two different versions of an abstract value domain module. The architecture of Angie is now fairly stable and reflects many hours of discussions regarding the overall design of the framework.

After this paragraph, chapter Preliminaries introduces the reader to the theory of abstract interpretation and symbolic memory graphs and provides an overview of *LLVM* project and the Predator tool. Also, it describes *SV-COMP* related experiments involving *Predator*. After that, a complete design of the framework is presented in an incremental way. Further, the current state of implementation is summarized, followed by a list of experiments performed with *Angie*. Finally, an outline of possible future development is given.

[4] See section 2.4 for more information about The LLVM Compiler Infrastructure

Chapter 2

Preliminaries

2.1 Symbolic Memory Graphs

The following chapter introduces the *Symbolic Memory Graphs* described in [7].

SMGs (*Symbolic Memory Graphs*) are a representation of memory — suitable for programs with pointers and linked-list structures — designed as an abstract domain for use within abstract interpretation analyses.

2.1.1 Abstract Interpretation

The following text is a slightly shortened version of abstract interpretation description given in [11].

> *Abstract interpretation* [5] is a theory of a sound approximation of the semantics of computer programs that, among other applications, allows for constructing static analyses sound by construction. Abstract interpretation consists in giving a class of programs a concrete and abstract semantics defined on suitable concrete and abstract lattice-based domains. These domains are usually linked by a pair of monotone functions—the so-called *abstraction* and *concretisation*, traditionally denoted α and γ, respectively—that form a Galois connection. Program statements are modelled as monotone functions, often called as concrete and abstract transformers[1], on the concrete and abstract domains, respectively.
>
> In a more general formulation of abstract interpretation [6], the requirement of dealing with a Galois connection is lifted, and the analysis is defined in terms of a concretisation (or, dually, abstraction) function only. This, however, excludes the possibility of defining best abstract transformers, which can be defined when using Galois connections (by simply first concretising the input abstract value, then using the concrete transformer, and finally abstracting the result). Another consequence of using the more general setting is that there is no easy way of comparing the precision of abstractions.
>
> In order to be able to use abstract interpretation for analysing programs, one further needs an operator for *accumulation* of abstract values[2] computed for

[1] In our work, a concrete transformer corresponds to an instruction and an abstract transformer corresponds to an `Operation`

[2] The operator in question is called join, usually denoted $\circ$

a single program point via multiple program paths. Moreover, since the abstract domain is often infinite, one needs the so-called *widening* operator ∇ that over-approximates the accumulation operator and that has the property that for any infinite sequence of abstract values $x_0, x_1, ...$, the sequence $y_0, y_1, ...$ where $y_0 = x_0$ and $y_{i+1} = y_i \nabla x_{i+1}$ eventually stabilises. The analysis is then performed by iterating the abstract transformers over the control flow graph, using the accumulation operator at program points where several program paths meet, and applying the widening operator at loop junctions to make the analysis terminate. Sometimes, the so-called *narrowing* operator Δ is also used after widening to refine its effect.

Moreover, for an informal introduction to abstract interpretation, the author of this thesis recommends web article called "Abstract Interpretation in a Nutshell"[3] and a summarizing work [4], both from the pen of the author of the original paper [5].

2.1.2 Elementary Graph Components

Figure 2.1 contains a simple graph illustrating the elementary components of an SMG, which are described in the following paragraphs.

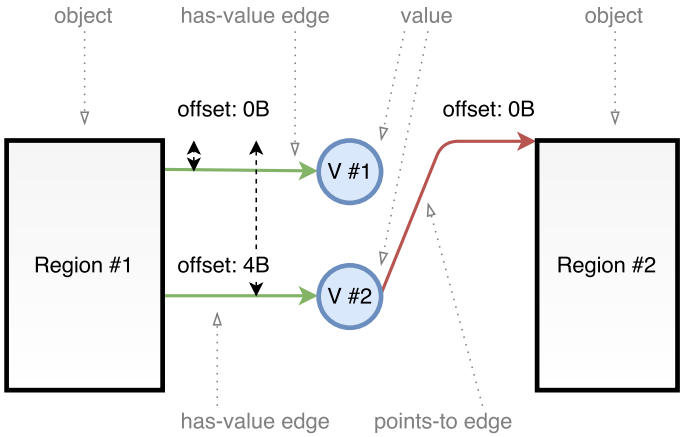

Figure 2.1:

SMGs consist of two primary kinds of nodes: *objects* and *values*. *Objects* are memory nodes which can be further divided into concrete *regions* and abstract *list segments*.

Moreover, *has-value edges* (one of the two types of edges existing in SMGs) connect *object nodes* to *values* and each such edge defines a field — a field represents a value that is stored in the source object at a given offset.

Values in SMGs are generally one of the unknown, integer and pointer types. Pointer *values* play an important role in SMGs, as they "connect" the Objects together — each pointer *value* node has one outgoing *points-to edge* pointing to an *object* with a given offset.

Nodes and edges in SMGs are labelled with more information then just a source or target offset, but those attributes were left out from Figure 2.1 for simplicity. An example is that of object size, field type or target specifier. For complete description of the graph attributes, please refer to sections 2.1 and 2.2 of [7].

[3] http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

2.1.3 Operations Defined Over SMGs

SMG is an abstract representation of a set of concrete memory configurations. Consequently, SMGs are also a part of abstract representations of sets of concrete program configurations, which must also describe the stack, registers, etc.

Based on that fact, SMGs must support abstract interpretation operations *entailment-checking*, *abstraction*, *widening* and *join* together with symbolic execution of memory-related program statements.

In case of SMGs, *entailment-checking* is implemented using the *join* operator.

Widening, which accelerates the potential convergence of states towards a *fixpoint*, is defined in terms of *abstraction* operator. *Abstraction* in SMGs tries to find uninterrupted sequences of *regions* forming a linked list satisfying certain conditions and turns them into one *list segment* node.

And finally, *join* which accumulates the abstract contexts into one, attempts to merge two graphs with the least possible amount of permitted changes.

Besides these operations, SMG has also a defined *read/write reinterpretation* operator, which can synthesize new fields and handles access to overlapping fields. Note, that any abstract object has to be concretized before being accessed.

2.1.4 Example of SMG with List Segment

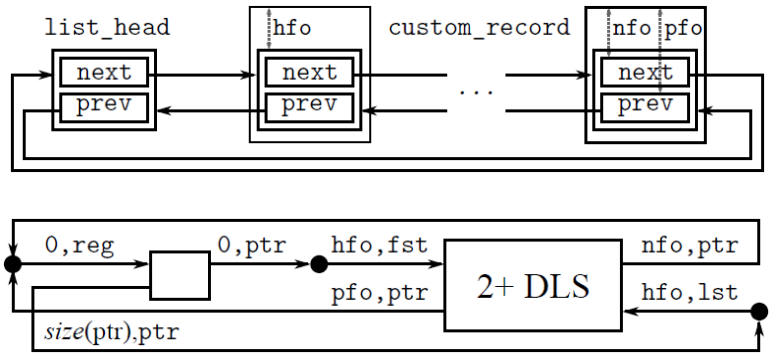

Figure 2.2: Standard memory view [top] and an abstract–SMG view [bottom]

Figure 2.2 shows[4] an example of *SMG* corresponding to linux-like double linked list. The list consists of head and no-less than 2 elements - first and last. Head of the list is represented by a *region* in the left part of the graph. It has two pointer value fields, one at offset 0, second at offset *size(ptr)*. Both corresponding points-to edges are pointing to the useful part of the list represented by 2+ DLS — the first and last element.

2.2 Predator

"Predator is a tool for automated formal verification of sequential C programs operating with pointers and linked lists" is the the description of Predator at its home-page[5].

Moreover, analysis performed by predator is sound and handles well all kinds of linked-list (circular, nested, and/or shared), address alignment, block operations, etc.

[4] First label of each edge denotes a source/target offset of *has-value*/*points-to* edges, respectively. Moreover, *hfo/nfo/pfo* stand for head/next-ptr/prev-ptr field offset

[5] http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/

2.2.1 Tool Overview

Predator itself has been through a lot of development, but its analysis is built on SMGs for quite some time. The structure of Predator could be divided into 3 main parts: the front-end comprising of a GCC compiler and CodeListener [8] adapter, the interpretation kernel with support for some of C Standard library functions, and the SMG algorithms.

The CodeListener infrastructure for operating over GCC (and possibly also other C-langauge compilers) internal representation is well tested, stable and is also used by another analysis tool called Forester.

Unfortunately, with regards to possible extensions of the tool, the interpretation kernel and SMG algorithms are closely coupled and the primary source of documentation is the source code with automatically generated Doxygen files, which makes getting to know Predator quite hard. Angie, like Predator, has a separate module for input pre-processing, but unlike it, the interpretation kernel of Angie is independent of the abstraction providers used in analyses.

As for other limitations, the most notable long-lasting weaknesses of Predator is its limited ability to treat non-pointer data.

2.2.2 Implementation

The implementation of SMGs in Predator does not match the description given in [7] exactly — there are some improvements that are only suggested in the paper, some algorithms are merged, and some technicalities are not covered in the paper at all. An example of differences we can give is that Predator supports concrete integers to a compile-configured boundary, provides a limited support for integer intervals and uses copy-on-write when creating derived SMGs. The latest versions of Predator also support emitting SV-COMP[6] error witness XML files.

Predator is implemented as a GCC plugin and its primary ouputs are GCC-formated warnings and files with graph plots. The simplest way to analyse a C source code file with Predator is to use a supplied script which invokes GCC with necessary parameters.

Predator comes with an extensive collection of test inputs, ranging from simple artificial examples and regression tests to a complex samples coming from complex applications. The author of this thesis have not found any particular C-language construct that would cause Predator crash during his experiments with the tool.

2.2.3 SV-COMP

SV-COMP[7] is a competition of software verification tools with goal to support and promote successful and stable verification tools. The tools competes in different categories and tries to verify relatively large sets of `C` programs. Both former and present VeriFIT members have contributed with various tools every year since the start of the competition.

2.2.4 Predator Hunting Party

Predator-HP[8] (Predator Hunting Party) is a modification of Predator developed originally for the 2015 edition of SV-COMP [12]. The motivation for creating Predator-HP was to

[6] Software Verification Competition

[7] https://sv-comp.sosy-lab.org/

[8] http://www.fit.vutbr.cz/research/groups/verifit/tools/predator-hp/

refine the final verdict of the tool and prevent the tool from producing unnecessary false alarms.

Predator-HP is effectively a bundle of several instances of Predator with different configurations. Whole Predator-HP is controlled by a set of shell and Python scripts and its latest configuration is one „verifier“, two „DFS Hunters“ and one „BFS hunter“.

- *prover* or *verifier* – Predator with all abstractions enabled, performing sound shape analysis. It could produce false alarms due to its use of abstraction and so the results are accepted only when it proves a program correct.
- *DFS hunters* – Predators reporting only errors, running without any list abstractions, but still with limited precision of arithmetic instructions and, most importantly, with different bounds on the depth of the state space search. For SV-COMP'15, there were 3 DFS hunters with limits of 400, 700 and 1000 GIMPLE[9] instructions. For SV-COMP'16 and '17, we used 2 DFS hunters with limits of 200 and 1000 GIMPLE instructions.
- *BFS hunter* – As for abstractions used, the BFS hunter is the same as the DFS hunter. Difference is that, while both the prover and the DFS hunter are performing DFS, BFS hunter obviously performs BFS which is more suitable for programs with deep yet narrow state space.

2.3 Predator-HP and SV-COMP Competition Contributions

The author of this thesis has participated in preparation and tuning of Predator-HP for SV-COMP'16 [9] as a part of getting to know Predator. The obtained configuration was however, used in SV-COMP'17 [10] without a major change.

We have created a set of scripts for running Predator-HP with all available test-cases and calculated overall score. Due to the penalization false-alarms, the score was almost halved compared to previous years. We identified approximately 10 problematic test-cases, discovered a weakness in the original shell script for translating output of Predator for the purpose of SV-COMP and optimized the Predator-HP configuration to improve results of Predator-HP for the competition.

Some of the test-cases triggered the prototype of interval abstraction in Predator, which caused a false alarm later during the analysis. We have been able to filter out most of these specific cases via a small modification of Predator which stops the analysis with *unknown* verification verdict. Unfortunately, this was not the case with the last two remaining false-alarm cases, which were also caused by sometimes faulty interval abstraction in Predator [2].

Modifications made to Predator-HP include:

- Predator now returns *unknown* result when performing operations over faulty interval abstraction domain,
- addition of new command line parameter to switch „uninitialized value“ between note and warning,
- addition of debugging code to discover maximum state space depth reached,
- consolidation of verifier/BFS/DFS code base.

[9] An instruction of intermediate representation used in GCC

2.3.1 Optimizing Predator-HP Configuration

The parameters chosen for SV-COMP'15 edition of Predator Hunting Party were rather experimental, and were questioned several times. We have decided to make a proper research regarding the optimal configurations of the tool for SV-COMP'16.

Based on the Predators configuration, correct-true results can be primarily returned by the *prover* or sometimes by the *BFS hunter*(when it finishes searching the whole state space without finding any problems), so there are not many options for tuning the tool for test-cases with no errors.

We have executed multiple runs of different predator configurations with all test-cases congaing an error, measuring the time and maximum reached depth. The set of Predators used was: prover, DFS hunters with depth limits granulary rising up to 2000, unbounded DFS hunter and BFS hunter.

During the experiments, we have discovered another weakness in Predator SV-COMP script that was greatly decreasing performance for longer running test-cases. After fixing it, we were able to terminate Predator up to 30% faster for long running tasks.

Based on the results, we have changed the composition of Predator-HP as mentioned earlier in the description of Predator-HP, and compared the new set-up of Predator-HP with the previous one using Benchexec[10].

2.3.2 The Analysis Results

Based on the benchmarking described above, we were able to draw several conclusions.

- In over 80% the program error can be found in the limit of 200 instructions. This makes DFS hunter with limit of 200 instructions a perfect candidate, capable of analysing many test-cases and saving computation time.
- In about 96% of cases, meaning all but four programs, the error can be found within the limit of 1000 instructions.
- The difference in run-time of DFS-900/1000 hunter and DFS-400 and DFS-700 hunters for the relevant cases is not big enough to cover up for the increased computation time consumption with more parallel hunters.
- In the remaining cases, the error might be much, much deeper in the state-space and so adding another DFS hunter with higher limit would not make any sense.
- There exist test-cases where DFS hunters return appropriate results, but BFS hunter does not - the state space of such test-cases is too broad.
- In some cases, the witness may be quite long, but the search space is narrow, so an error can still be found by the BFS hunter

There were also some programs both proved correct by the verifier but not the BFS hunter and proved correct by the BFS hunter but not the verifier.

Figure 2.3 shows how the different configurations of Predator were able to cope with different test-cases.

[10] A tool for reliable benchmarking of verification tools used in SV-COMP https://github.com/sosy-lab/benchexec

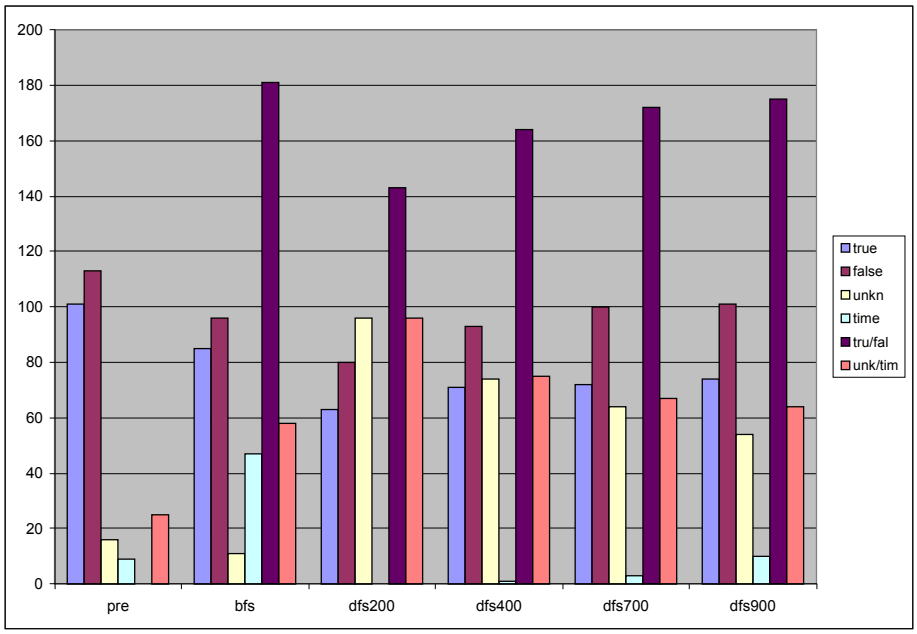

Figure 2.3: Distribution of true/false/unknown/timeout results for different configurations of Predator

2.3.3 Summary and Results of SV-COMP 2016

The final results of the SV-COMP'16 are published on the competition website http://sv-comp.sosy-lab.org/2016/results/results-verified/ [2]. After the reorganization of categories which happened before SV-COMP'16, Predator-HP attended the competition only in the category „Heap Data Structures". Results for the category were:

1. PredatorHP with 298 points and 1100s of cpu time.
2. CPA-Seq with 234 points and 4100s of cpu time.
3. Cascade with 197 points and 9900s of cpu time.

The maximum score for the category was 382 points. The score calculation description is available online at http://sv-comp.sosy-lab.org/2016/rules.php. We have not only managed to repeat our victory in „Heap" catogry from previous years but also to improve our total computation time in the competition. Described optimization technique can be used for example with automatic verification in continuous/regress testing scenarios.

For completeness, we can note that Predator-HP won a gold medal in SV-COMP'17 too.

2.4 LLVM

In this section, we will first cover the LLVM project, its purpose and some of the tools of the project and then give a description of the LLVM intermediate representation (IR): its representations, specifics of SSA-based[11] form and list of important instructions.

2.4.1 LLVM Compiler Infrastructure Project

The LLVM compiler infrastructure project "is a collection of modular and reusable compiler and toolchain technologies" [http://llvm.org/] written mainly in C++.

The LLVM Core module defines and works with intermediate code representation known as LLVM IR and provides both target-independent optimizations and target-specific code-generators.

Clang — defined in a context of the LLVM project — is than a swiss-army-knife style compiler front-end, focused not only on fast compilation but also on reusability of its components. Clang can be used to built a static analyser[12], to power an IDE auto-completion and documentation features[13,14,15], to check validity of doxygen comments[16] , and many more.

When used as a C/C++ compiler in a standard command line mode, clang driver module comes into place, serving as an umbrella for both libclang and llvm libraries and controls the compilation, optimizations and code-generation phases. Clang can be set to compile not to native code but only to LLVM code - more about LLVM as a language in next subsection.

LLVM binary distribution also contains other usefull tools, some of those are:

- opt - transforms LLVM code using desired transformation passes
- llc - compiles LLVM code into native machine code
- lldbg - LLVM native debugger

The LLVM project major version work cycle is 6 months long, with no guarantees of C++ API stability between major version changes. There is also a more stable C API, used for bindings to other programing languages (for example Python). The source code is available under a permissive software license.

2.4.2 LLVM as a Language and the Intermediate Representation

LLVM also stands for the LLVM assembly language, also known as LLVM assembly, LLVM IR language, LLVM bitcode or sometimes anachronistically LLVM bytecode. LLVM, in this context, is a Static Single Assignment based representation of a computer code, suitable for optimizations [1]. LLVM can take three forms in the LLVM toolchain:

- object form in the memory of LLVM family tool which can manipulated using an API, called an LLVM IR or just IR
- textual form / assembly language representation, as described in the language reference [1], typically using an .ll file extensions
- Binary form, called a LLVM bitcode[17], typically using an .bc file extension

[11] Static Single Assignment
[12] http://clang-analyzer.llvm.org/
[13] http://codelite.org/LiteEditor/ClangIntegration41#toc2
[14] https://github.com/Rip-Rip/clang_complete
[15] https://github.com/justmao945/vim-clang
[16] http://llvm.org/devmtg/2012-11/Gribenko_CommentParsing.pdf
[17] http://llvm.org/docs/BitCodeFormat.html#overview

Quote from [1], Introduction section:

> It aims to be a "universal IR" of sorts, by being at a low enough level that high-level ideas may be cleanly mapped to it (similar to how microprocessors are "universal IR's", allowing many source languages to be mapped to them).

For a list of possible reasons, why prefer LLVM infrastructure, over (for example) GCC, the author suggests to read the article [13].

We will further refer to both the LLVM as an assembly-like language and LLVM as an in-memory IR as to LLVM IR. That will allows us to distinguish the language from LLVM as a set of libraries.

2.4.3 Important Concepts and instructions of LLVM IR

We will now describe the basic concepts and instructions of LLVM IR.

Memory Access and Allocation

The only instructions operating directly on memory are `load`/`store` `load` and `store` instruction. Operands of these instructions must by of first-class type or pointers to such a values. First-class types are essentially all types except for void, opaque types and target(machine)-specific types.

The only memory allocation instruction directly known to LLVM is `alloca` , operating on function's stack frame. Front-ends usually generate one `alloca` for every local automatic variable.

Most `alloca`s representing local automatic variables can be optimized out together with all their connected stores and loads, emitting `phi` instructions in case the memory was stored to on multiple paths in the control flow graph (`phi` is an instruction specific to SSA form and is explained in the **Control Instructions** block). Note that the code containing stores and loads is only in a partial SSA form[18]. This process is also called lowering of memory to registers.

The `getelementptr` instruction — shortened as GEP — computes the pointer to an element of an aggregate using the base pointer and N indices to the aggregate type, as explained on the example below. Note that when an aggregate variable is transformed from memory to register using the aforementioned lowering process, GEP instructions preceding the loads and stores are replaced with `extractelement` and `insertelement` instructions.

GEP is quite complex and has its own documentation webpage[19] in the LLVM project docs, where more details can be found. We now present a simple example of load/store and GEP addressing generated for struct member access in Figure 2.4.

Integer Arithmetic

LLVM IR instructions covering integer arithmetic include two's complement integer arithmetic instructions `add`/`sub`/`mul` and the signed/unsigned variants of instructions computing quotient and remainder `sdiv`/`udiv`/`srem`/`urem`.

[18] A code that does not use only SSA registers but also accesses the memory is no longer really „single assignment" as memory can we written-to repeatedly. Such a code is sometimes refereed to as code in a partial SSA form

[19] http://llvm.org/docs/GetElementPtr.html

```
struct Example {
    int16_t elem1;
    int32_t elem2;
};

...
    struct Example stackvar;
    stackvar.elem2

                    = 42;
...
```

```
%stackvar = alloca %struct.Example
%tempptr  = getelementptr %struct.Example,
            %struct.Example* %stackvar, i64 0, i32 1
store i32 42, i32* %tempptr
```

Figure 2.4: Excerpt of code: memory access and allocation in C and LLVM IR, side by side

The integer representation of LLVM type system is signless as opposed to the C-language family type systems, and uses the two's complement modulo arithmetic integer representation. To express the different behaviour of sign/unsigned types in the C-language family (and other languages), LLVM IR has a set of instruction flags/keywords stating whether a singed or unsigned overflow of an integer operation leads to an undefined value.

Note that the C standard defines unsigned arithmetic in terms of two's complement modular arithmetic, which is the arithmetic model used in LLVM IR, so no flags like those mentioned above are generated for these operations.

Further, the `and`/`or`/`xor` instructions provide standard bitwise operations in LLVM IR. Shift instructions produce undefined values if the number of bits shifted is equal to or larger than the size of shifted type. The left shift supports the same overflow flags as do the basic integer arithmetic instructions, while the right arithmetic and logical shift do not.

Control Instructions

We start with the `br` — a branching instruction that can be either unconditional — has one target, it is effectively a jump — or conditional — has two targets. The type of the control value is a one-bit wide integer. A switch is then a generalization of the `br` instruction and is defined in terms of value-target pairs (4, blockA; 5, blockB) where the type of all values in pairs and the control value must match.

Subprogram control instructions, i.e. mainly `call`, `ret` (and also `invoke` and others, when exception handling is supported), are following the usual semantics — the control is transferred onto the called code and returned to the following instruction in the calling code. The type of the return value of the call instruction matches the signature of the called function.

The operand of the `ret` instruction is linked to the resulting value of the calling instruction upon return. Call arguments are linked to the formal parameters of the function being called upon the call in the same way.

Finally, `phi` is an instruction specific to the SSA form and allows elimination of memory allocation and access while retaining the option of having multiple paths in the program affecting the same program variable. The result of a `phi` instruction call is one of its value operands, selected based on the previous program location on the current execution path. For more details, see [20]. `phi` instructions cannot be realized directly on most of the current hardware, and their behaviour is typically replicated by usage of the same physical register or the same memory location in all paths of the program, or using a switch-like instruction.

[20] https://en.wikipedia.org/wiki/Static_single_assignment_form

Conversion Instructions

Conversion instructions can be divided into two categories based on their effect on the underlaying data: no-ops and data-conversions.

No-op cast instructions change only the type of operands and include, e.g., conversions from pointers to integers (of the same bit width) or pointer casts.

Data-converting instructions cover integer to floating pointer conversions, integer truncations and extensions, etc.

Others

Other LLVM IR instructions include floating point instructions — these generally reflect the design of their integer counterparts. As LLVM IR is also modelling some higher-level language constructions, it also provides instructions for thread-safety (atomic operations, memory barriers,...), exception handling (which is conceptually different on different ABIs), various intrinsic functions, flags and metadata kinds.

Finally, LLVM IR supports usage of `constexpr` — constant value expressions — as operands of instructions. Such an expressions are usually generated during optimizations to make the resulting IR code smaller, but they can easily unwrapped into separate instructions to simplify the instruction set.

2.5 Other tools and frameworks

This sections lists some of the existing projects devoted to development of software verification tools and frameworks.

- *CPAchecker*[21] is a platform for software verification written in Java and based on the idea of Configurable Program Analysis (CPA). It allows for composition of analysis and one of the supported analyses is even a simplified version of the analysis implemented in Predator (so far without a support for abstraction). Unfortunately, CPAchecker forces developers of the analyses to be used within it to accept the approach of CPA which may sometimes be restricting.

- *Ultimate*[22] is also written in Java, uses a dialect of *Boogie*[23] for intermediate representation, and concentrates on analyses implemented in
an automata-theoretic way. The latter is again somewhat restricting.

- *Frama-C*[24] is one of the most known *plug-in* based platform for C analysers, it operates on *CIL*—the *C Intermediate Language*[25] and is written in OCaml as many others research verification tools. Here, the use of CIL is not much aligned with the latest development in the world of major compilers such as those of LLVM[26] family.

- *Infer abstract interpretation framework*[27] is a recent addition to the *Infer* static analysis tool, originally intended for bi-adductive analysis of programs with dynamic data

[21] https://cpachecker.sosy-lab.org/
[22] http://monteverdi.informatik.uni-freiburg.de/ultimate/
[23] https://github.com/boogie-org/boogie
[24] https://frama-c.com/
[25] https://people.eecs.berkeley.edu/~necula/cil/
[26] http://llvm.org/
[27] http://fbinfer.com/docs/absint-framework.html

structures based on separation logic. This framework could be an alternative option for implementing a new SMG-based tool. However, after discussions in the VertiFIT group, it was decided to have our tool actively under the control of the group. Like *Frama-C*, *Infer* is implemented in OCaml.

In our framework, we would like to rely on a major compiler infrastructure, and we would like to be as little restricting in terms of the kind of analyses supported as possible.

Because we decided to build on LLVM, we would like to mention two further recent tools that also use the *LLVM* tool-chain as their front-end and also compete in *SV-COMP*:

- *Symbiotic*[28] is a verifier based on symbolic execution written in C++.
- *DiVinE*[29] is also written in C++, focuses on verification of concurrent programs, features a virtual machine interpreting the *LLVM bitcode*, debugger displaying the program in *LLVM assembly language* and others.

None of the above two tools, however, can be viewed as an infrastructure.

[28] https://github.com/staticafi/symbiotic/
[29] https://divine.fi.muni.cz/

Chapter 3

Towards Framework for Static Analysis

This chapter presents the design of Angie framework from a top-down perspective.

We first identify the three main phases of a program analysis process and place the key Angie parts into the individual analysis phases. As the next step, we introduce the reader to our concept of an abstract interpretation loop.

After that, we give a detailed description of Angie in three bigger blocks.

- *Input Program Pre-processing* covers the transformation of C source code to an Angie internal representation.
- *ValueContainer* demonstrates an Angie component for representation of values in abstract domains.
- *Analysis Itself* focuses on interpretation-related parts of Angie and presents the final abstract interpretation loop.

A Note on Colour Code Used in Diagrams

Unless stated otherwise, figures used in this chapter use the following colour coding to describe the level of dependency of framework component on user-supplied (e.g. provided by a user of the framework) code.

- Green means no dependency on user-supplied code
- Yellow means that component depends on a user-defined type, meaning that such a component must be able to accept polymorphic objects. That dependency can be either reference to an object of such type or an instance of the object directly.
- Orange marks a user supplied component realizing an interface defined in framework.
- Red stands for completely user-defined and framework-independent code.

In case the component is not a part of framework code but an accompanying script or application, the semantics is yellow for “configured by the tool” and red for ”fully independent”.

3.1 Some Introductory Notes

3.1.1 Simplified View of Static Analysis Tool

The view of the static analysis tool structure can be simplified to three components, as shown in the Figure 3.1.

Figure 3.1: Simplified view of static analysis tool.

Regardless of the form of the analysed program, is is usually not suitable for direct analysis. Because of that, the first step is conversion to a designated intermediate representation. After that, one or more analyses sequentially are run, passing results from one to another. Some analyses also do a pre-run of auxiliary analyses for the purpose of code annotation or further transformation. Both during and after the analysis, results are serialized into the desired format, which could be a structured error trace, invariant annotations of input program or a plain textual log.

3.1.2 Basic Abstract Interpretation Analysis Loop

Angie is a static analysis framework, designed primarily to aid implementing abstract interpretation analyses of C programs. Algorithm 1 shows — a slightly rephrased abstract interpretation loop, which is written in a more imperative view than often viewed and that is a conceptual base for analysis implemented in Angie.

3.1.3 Important Components of Angie Framework

Analysis built using the Angie framework then relies on the following important components: `CfgNode`, `Operation`, `ValueContainer`, `State` and `StateManager`. Their place in the the basic tool model can be seen in Figure 3.2 and their data dependency diagram is shown for illustration on Figure 3.3.

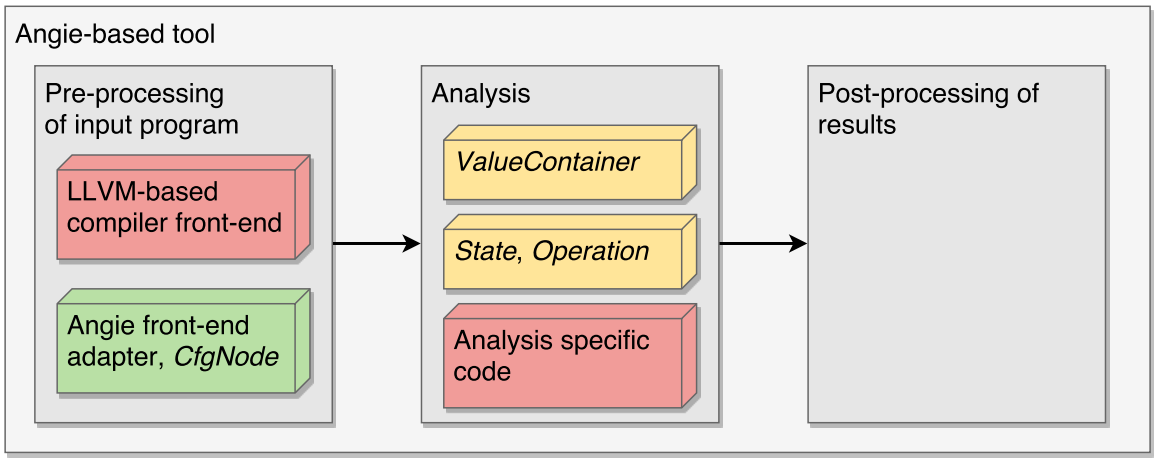

Figure 3.2: Static analysis tool, Angie-based.

Input: CFG
Output: labelled CFG where each location is labelled by a set of abstract states that overapproximate the concrete states reachable in the CFG
Data: `worklist` of abstract states. Abstract state is also called a symbolic configuration of program, constisting of symbolic state of memory, registers and program counter (pointing to the next program location)
Result: abstract state space of the analysed program has ben explored

```
1  Initialize worklist by inserting initial state, tied to the first program location;
2  while Worklist is not empty do
3  |  Fetch and remove an abstract state s1 from worklist;
4  |  Compute the set S2 of successors of s1;
5  |  foreach s2 ∈ S2 do
6  |  |  Let S be the set of abstract states curently associated with location s2.loc ;
7  |  |  if s2 is not covered(entailed) by any abstract state curently in S then
8  |  |  |  Add s2 to S possibly joining it with some element(s) already in S,
   |  |  |   possibly applying widening/abstraction too;
9  |  |  |  Add s2 (or emelement obtained from it by abstraction/widening/join)
   |  |  |   into the worklist and eliminate from worklist elements covered by the
   |  |  |   newly added state;
10 |  |  end
11 |  end
12 end
```

Algorithm 1: Abstract interpretation analysis loop

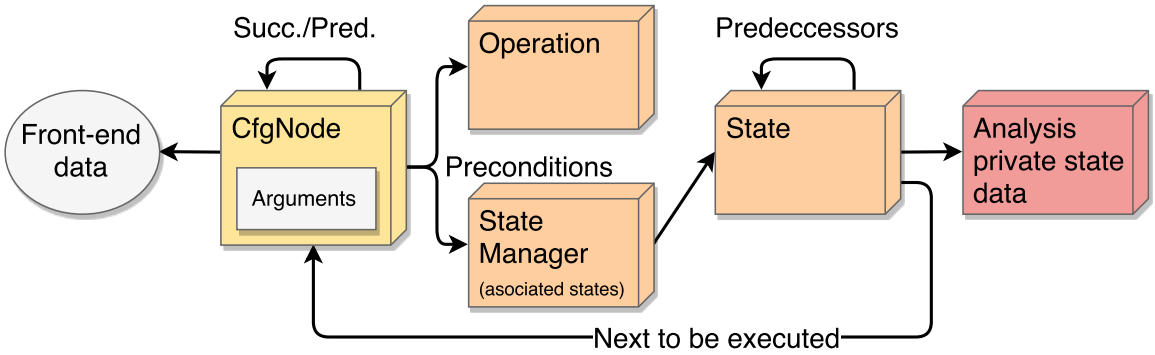

Figure 3.3: Data dependency diagram of important analysis components.

`CfgNode` is an Angie variant of program statement (in fact, CfgNode is more granular and corresponds to single IR statement), `Operation` is then an abstract counterpart of instruction type (binary arithmetic instruction vs subtraction). These two will be described more in the following section.

`ValueContainer` is designed to represent a set of values in a certain abstract domain (like integer intervals or polyhedra), `State` is an abstract program state and `StateManager` is black-box that serves as list of CfgNode-associated states and a smart global worklist (it triggers entailment checking, join and/or widening operations when necessary).

3.2 Input Program Pre-processing

Pre-processing of the input program consists of two main steps:

- a transformation of the input program into an intermediate representation and
- a pre-analysis which labels the intermediate representation with some auxiliary information such as liveness of variables.

While implementation of those two steps is generally re-usable across a wider range of analyses, for some analyses, it might be beneficial or even necessary to further pre-process the intermediate representation in a way more tailored to the given analysis.

Examples of such analysis-specific pre-processing steps are:

- inserting meta-instructions, like abstraction or widening, to accelerate the execution of loops (so analysis does not have to detected loop-points repeatedly),
- replacing call instructions for a set of (standard library) functions by special instructions (like malloc, free, floor, ...) to make the analysis structure clearer and for performance reasons too (prevent repeated string queries when comparing the function name with a list of known external functions),
- removing unnecessary instructions, e.g. instructions ignored by some analysis.

Angie uses an LLVM-based compiler front-end to first compile the source code of the input program into the LLVM intermediate representation (further referred to as LLVM IR), which is then transformed into the Angie intermediate representation (Angie IR). Angie IR retains many of the LLVM IR properties and acts as a thin wrapper insulating the analysis developer from changes of LLVM breaking even its API. Note however, that although Angie IR is highly influenced by the LLVM IR, other front-ends besides LLVM-based ones could be used, provided that an appropriate front-end module emitting Angie IR is supplied for them.

As mentioned before, the compilation into the LLVM IR is carried out by an external component (compiler), therefore it is not further elaborated in this chapter. Now, we will first introduce the notions of the `Operation` concept and the `CfgNode` component that we will use to represent CFGs. Later, we will provide a description of LLVM and Angie IR. Finally, we will describe our algorithm for transforming LLVM IR into Angie IR.

3.2.1 Operation

`Operation` in Angie denotes a function which

- takes zero or more arguments,
- can be applied to an abstract state of a program — which is referred to simply as a State in this work,
- results in a set of new `States`[1].

Note that while a word *instruction* in the context of LLVM IR refers to a concrete instance of a certain instruction type (at a specific location in a program, with a set of assigned arguments), an Angie `Operation` is a loosely specified abstract counterpart to a set of instruction kinds (`Operation` „Binary Arithmetic Operation“ is a counterpart to LLVM instructions like `add`, `sub`, ...)

[1] Although it might not be obvious, it is generally possible for one of the resulting states to be completely equal to the original one, for example with an infinite single-instruction loop. `Operation` forbids this, and resulting states has to differ from the source one — such a difference could be a number of generation of the state or a list of predecessors.

3.2.2 LLVM IR as a CFG

The left part of Figure 3.4 shows a simplified excerpt of LLVM IR in the form of traditional control flow graph (CFG): the code is organised into so called basic blocks with edges leading in only to the first instruction of the block and leading out only from the last instruction of the block. The entry and exit blocks are exceptions since they have only outgoing or incoming edges. The right part of the figure illustrates the role of `CfgNode` and is further elaborated on in the following subsection.

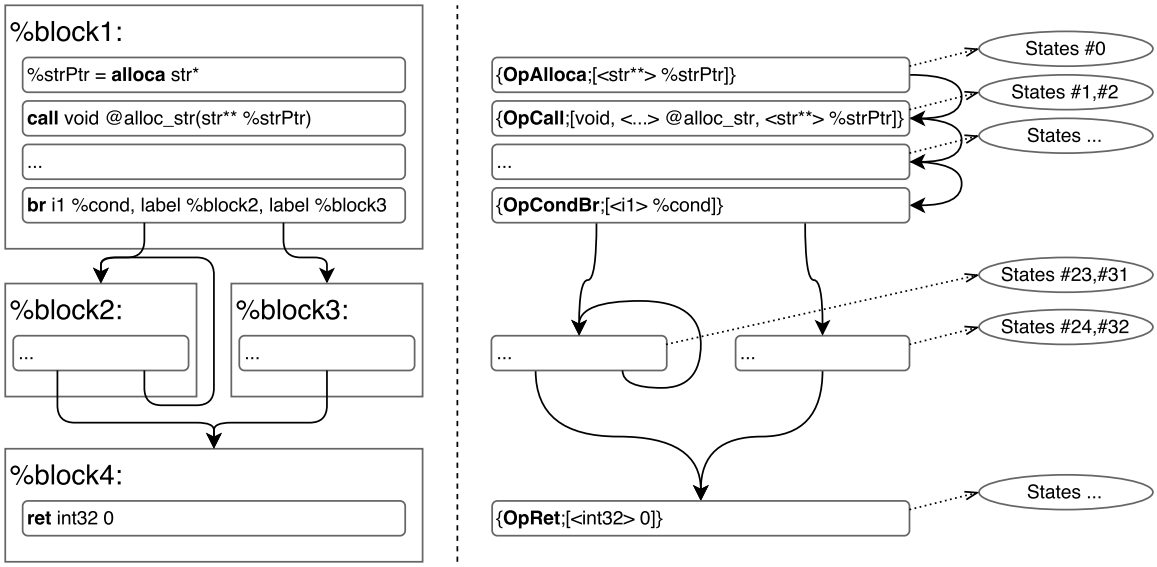

Figure 3.4: An excerpt of LLVM IR in the form of traditional CFG (left) and the corresponding simplified CFG consisting of `CfgNode`s (right).

Not all analysis, however, benefit from the usage of basic blocks. That is why, in Angie, we decided to simplify the representation and define CFGs on the level of particular operations. For the same reasons, we support only at most two node successors in our graph, eliminating `switch` instruction. For the rest of the thesis, we will refer to this simplified form of CFG as simply CFG.

3.2.3 CfgNode

We define a `CfgNode` as a node of our CFG, effectively representing program location at the level of Angie IR. Every instruction in the front-end IR is mapped to one `CfgNode` and vice versa — in other words, we set a bijection between the front-end IR and Angie IR on the level of instructions and `CfgNode`s, respectively.

Figure 3.4 demonstrates the transformation: for every basic block b within front-end IR, every outgoing edge e_1 is transformed for Angie IR into edge e_2 leading from last `CfgNode` c_{last} created from the block b to first `CfgNode` c_{first} created from the block e_1.target.

To ease the navigation in the graph, we also define backlinks: links to all predecessors of the node (these are not shown on Figure 3.4).

The right side of Figure 3.5 shows a data dependency diagram of `CfgNode`, where all the important Angie components are depicted as separate blocks. `CfgNode` is then a tuple consisting of an `Operation`, a list of arguments, a list of `State`s found that are reachable at this `CfgNode` (as described in Algorithm 1), a list of successor nodes and a list of predecessor

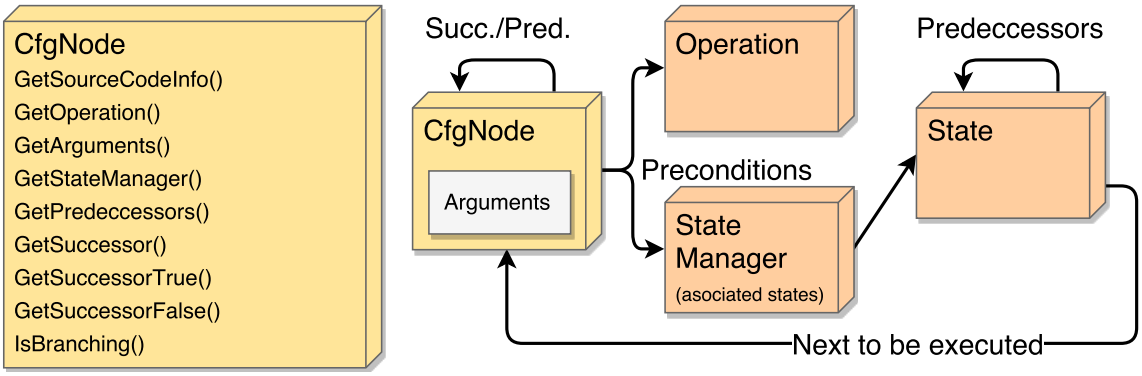

Figure 3.5: A list of properties of `CfgNode` component (left) and a data dependency diagram of `CfgNode` (right).

nodes. Also, note that the association between `CfgNode`s and `State`s is circular — a `CfgNode` has its associated `State`s, and a `State` has its one associated `CfgNode`.

The right side of Figure 3.4 shows a CFG consisting solely of CfgNodes, corresponding to the original LLVM IR CFG on the left side. Each node is depicted as a tuple of an `Operation` (bold) and transformed instruction arguments (inside square brackets). The types of arguments are specified in the angle brackets. The connection of each CfgNode and its associated States is implemented by a `StateManager` component (described more in further sections) in the proposed design. However it is simplified in the figure to a direct dotted line, placed in the right-most section.

The left side of Figure 3.5 shows a list of methods of a `CfgNode` component — the implementation of these is not important, but we except them to be simple get accessors. *GetOperation*, *GetArguments*, *GetStateManager* retrieve the data associated with the location represented by the `CfgNode`, *GetPredecessors* all predecessor nodes, *GetSuccessor* retrieves the single successor node if the node is non-branching and its *GetSuccessorTrue/False* counterparts do the same for branching nodes. *GetSourceCodeInfo* returns an object that can be used to query information about the related front-end IR and source code (source code file name, row, column, etc.).

3.2.4 FrontendId

Apart from constant and global values, all values in LLVM IR are identified by the instructions that created them.[2] During our transformation from the LLVM IR to the Angie IR (described in the next subsection), each encountered non-void instruction as well as all operands of all instructions (void or not) are assigned an opaque unique identifier[3] called `FrontendId`, to be used in the Angie IR. Note that instruction operands in LLVM can be of many kinds, not only values, but also types, metadata, etc. Hoewever, we only assign `FrontendId`s to values and — as a special exception — to functions. Nevertheless,

[2] In fact, in LLVM C++ design, identity/equivalence of values is based on object (pointer) equality. Also, the Instruction class is a subclass of Value and its instances are used directly as operands. It is thus impossible to separate the concept of instruction's return value and the instruction itself, because they are the same object in LLVM.

[3] Such an identifier can be obtained for example (i) from the pointer identity of the original value or (ii) using a map between LLVM API objects and assigned Angie IR FrontendIds

only a small set of `Operations` (in particular `call`, `invoke`) use these specially crafted `FrontendIds` as arguments.

3.2.5 The Transformation Process

The Diagram in Figure 3.6 outlines the process of front-end IR parsing—so called Angie front-end adapter. The parser is responsible for transformation of the input front-end IR (in our case LLVM) into Angie structures — e.g. it constructs the Angie IR.

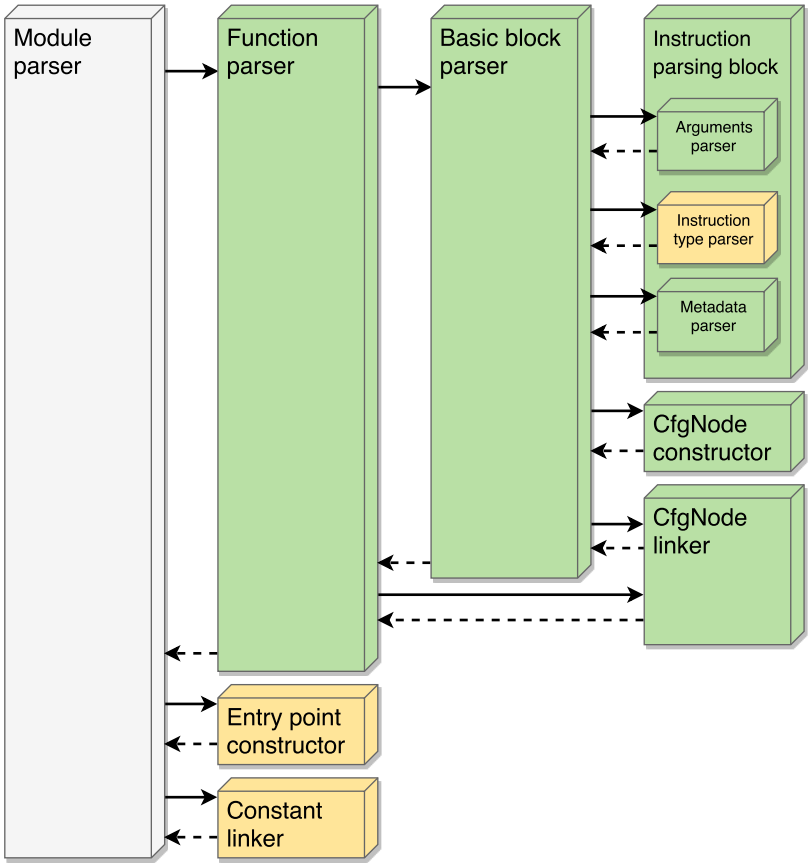

Figure 3.6: „Call graph“ of the front-end adapter

Apart from modules, functions, basic blocks, and instructions parsers, where the names are suggesting their basic functionality, there are also the **CfgNode constructor** and **CfgNode linker**. The former uses information obtained from the **Instruction type parser** to construct a new `CfgNode`. The former then deals with linking the `CfgNodes` together and manages the bijective mapping between basic blocks and `CfgNodes`[4].

The parsing starts on the level of a translation unit—a module. A module is generally a set of functions and their accompanying meta-data (the visibility, entry-point, ...).

Every function f in a module is processed by the **Function parser** which passes the entry basic block and all remaining blocks of f to **Basic block parser**. The **Function parser** also creates a `FunctionHandle` object for every function and inserts it into map of functions called `FunctionMapper` (described more in analysis Subsection).

[4]More precisely, between each basic block b and the `CfgNode` c created for the entry instruction of b

The **Basic block parser** sequentially processes the instructions in the block using the interconnected **Instruction type parser**, **Arguments parser**, and the **Metadata parser** of **Instruction parsing block**, and then it returns the entry `CfgNode` generated for the block.

Before we look more in-depth into the **Instruction parsing block**, we will now describe the operational principle of the whole parser. The **Function parser** together with the **Basic block parser** perform a breadth-first traversal[5] of the input CFG consisting of basic blocks. Whenever a second basic block of a neighbouring pair is parsed, their respective exit and entry `CfgNode`s are linked together (as described and visualised in Section 3.2.3).

Now, let us go back to the **Instruction parsing block**. The role of the **Instruction type parser** is to produce an appropriate `Operation` based on the type front-end instruction. Every analysis has to supply an `OperationFactory` which provides an appropriate implementation of the selected `Operation`.

The **Arguments parser** then processes the instruction (again) together with its operands/arguments and turns them into a set of arguments and/or special flags for the selected `Operation`. For example, a binary arithmetic operation, `BinOp`, has two flag fields: one specifying the particular arithmetic operation and one controlling signed/unsigned and overflow/underflow modes. Also, if a constant operand is encountered, it is queued for later processing in the **Constant linker**.

The **Metadata parser** reads basic block boundaries, debug info, and other metadata from front-end IR. Note that a special metadata trap has been implemented for the Angie prototype, so that any marked LLVM IR instruction can switch the tool to debugger upon interpretation — more information can be found in Section 4.7.

After our CFG has been finalized, the initial State for the analysis has to be constructed. Since the description of most analysis-related components is given in the following sections, we only describe OperationFactory and ValueMapper now.

OperationFactory is an abstract-factory[6] for creating `Operation`s implemented for a particular analysis. Since it allows to create all possible `Operation`s, its list of methods is too long for direct inclusion.

`ValueMapper` provides a mapping from `FrontendId`s of SSA registers to `ValueId`s.

The problem is that *main*'s formal parameters or globals in commonly generated LLVM IR do not have any point of definition in the CFG — they are only turned into `FrontendId`s during the aforementioned parts of the transformation. Consequently, when an analysis would encounter these FrontendIds in place of arguments, there would be no values assigned to them and a fatal error would follow (another example of breaking the SSA form of the code).

To solve this problem, we introduce the **Entry point constructor** and **Constant linker** to construct an initial `State` for the analysis and assign values to these problematic `FrontendId`s.

In particular the **Entry point constructor** creates a dummy entry point function that (i) provides values for `main`'s formal parameters and (ii) initializes module's static data.

The **Constant linker** then parses the LLVM IR representation of constants and globals and inserts it into the `ValueContainer` in the initial `State`.

[5] Breadth-first traversal is not the only option of LLVM CFG traversal, but is the most generic one. The straightforward way would be to sequentially process all blocks of a function using an LLVM iterator and to let the **CfgNode linker** handle the rest.

[6] Abstract-factory is a design pattern

3.3 ValueContainer

`ValueContainer` is a component for storing and manipulating representations of abstract values used during an analysis. We will demonstrate the properties of `ValueContainer` on a series of examples and sumup the important information at the end of the section.

The examples are conceived such that we present a C code fragments assumed to be analysed, followed by a C++ code fragment corresponding to code that an analyser in Angie would perform when analysing the code.

3.3.1 Variables vs. Values and the ValueId

While C variables are repeatedly assignable, values are immutable — once a certain abstract value has been obtained, its representation can be refined, but the value itself cannot be changed. We will illustrate the concept of values on the short code fragment in Algorithm 2 that consist of a variable definition and two assignments to that variable.

```
1    int32_t myVar; // definition of variable
2    myVar = 4;     // assigned
3    myVar = 6;     // re-assigned
```

Algorithm 2: A C fragment with a variable being re-assigned

Now, the C++ pseudo-code in Algorithm 3 creates two independent values as if we try to interpret the preceding C code fragment within Angie. We explain the code in more detail bellow.

```
1    ValueContainer vc; // initialize new ValueContainer
2    ValueId myVar4 = vc.CreateConstantValue(Integer, size: 32b, value: 4);
3    ValueId myVar6 = vc.CreateConstantValue(Integer, size: 32b, value: 6);
```

Algorithm 3: `ValueContainer`: values are immutable

After an initialization on line 1, a representation for a 32 bit constant integer of value 4 is created inside the `ValueContainer` on line 2. Also, a `ValueId` is returned from the call — it identifies the created value across the calls to `ValueContainers`[7] (an alternative name for it would be a value handle). After that, a value of the same type representing a constant of 6 is created on line 3. At this point, the analyser created two independent values which it could one-by-one assigned to a single memory location representing a variable to complete the interpretation.

3.3.2 Unknown Value and its Refinement

We now continue with an example of C source code shown in Algorithm 4 containing a conditional statement and analyse it from the point of view of variables and values.

Variable x on line 1 is volatile, which means that every read emits a potentially different value. Variable $fixed\_x$ on line 2 contains a „freezed“ value of x at the time of its read on line 2. From the analyser point of view, the concrete semantics of this value is unknown.

[7] One value can have multiple representations, one per an existing `ValueContainer`, and thus the `ValueId` must be a globally unique identifier. More about that later in this sec.

```
1  volatile int32_t x;
2  int32_t fixed_x = x; // obtain an unknown value
3  if (fixed_x >= 10) {
4     // point A, value is known to be >= 10
5  } else {
6     // point B, value is known to be < 10
7  }
```

Algorithm 4: An example of C source code with an unknown value which can be refined during an analysis

Because the values are of type int32_t[8], it can be represented in the integer interval domain as $[-2^{31}, 2^{31} - 1]$. The program condition based on an integer comparison on line 3 leads to a branching of the execution. As a result, the integer interval representation of the value assigned to $fixed\_x$ can be refined to $[10, 2^{31} - 1]$ at point A, and to $[-2^{31}, 9]$ at point B.

An analysis run of the preceding C code would correspond to the C++ pseudo-code in Algorithm 5 which constructs and refines an unknown value accordingly.

```
1  IntervalValueContainer vc; // initialize new IntervalValueContainer
2  ValueId fixed_x = vc.CreateValue(Integer, size: 32b);
3  ValueId const10 = vc.CreateConstantValue(Integer, size: 32b, value: 10);
4  // if (!(vc.IsCmp(fixed_x, const10, CmpFlags::Lesser) == tribool::true))
5  vc.Assume(fixed_x, const10, CmpFlags::GreaterOrEqual);
```

Algorithm 5: `ValueContainer`: unknown value and its refinement

A default representation for a 32 bit unknown integer value is created inside the `ValueContainer` on line 2. After that, a constant value of 10 is created on line 3. Finally, the value $fixed\_x$ is refined[9] on line 5 – as if the control of program reached the line 4 of the C-code in the Algorithm 4 (point A).

Note that this (and all the other) pseudo-code only focuses on the aspects of `ValueContainer` — in the case of full analysis attempting to reach the point A, a tribool[10] query similar to that placed in comment on line 5 would have been executed against the `ValueContainer`.

A Note on Terms Regarding Unknown Values

Although the terms unknown value, indeterminate value, undef and others come from different backgrounds, they refer to a similar concept: an abstract value originating from an undefined behaviour[11]. However, they are understood differently in their respective backgrounds[12].

For example, in LLVM, the established "undef" can yield a different value upon every read – a property that has both advantages and disadvantages for analyses and optimizations. To broaden the possibilities of the compiler, the community invented "poison" values

[8] Assuming that the architecture supports integer at least 32 bits wide.

[9] shortened for readability, the full form would be: the value identified by the `ValueId` $fixed\_x$ represented in the `ValueContainer` vc is refined.

[10] tribool is a type capable of holding values true, false, indeterminate

[11] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

[12] http://sunfishcode.github.io/blog/2014/07/14/undef-introduction.html

with a different semantics[13]. To further complicate the matter, LLVM is a live project and there are ongoing discussions about what is the ideal variant of the „unknown value“ concept to be used in the LLVM IR[14].

For now, it suffices to say that unknown value in the context of `ValueContainer` behaves as any other value: once created, it can not be changed, just refined when one knows more about it during program execution. This is, as we have seen, an unknown integer can be verified to an integer ≥ 10 and later, e.g., to an integer $\in [10, 10]$. Moreover, once we test it, e.g., to be equal to 5, it will stay 5 for the further execution of the analysed program (unlike the LLVM “undef” which corresponds more to an unknown value generator).

3.3.3 Operations over Values

So far, we have obtained values in two ways: (i) creating a new unknown value without any information except for its type or (ii) creating a new value using a constant, again, with a given type. However, we usually use some existing values as operands to an operation to obtain a new value as a result. The code in Algorithm 6 shows an example sequence of operations which can be immediately evaluated to a constant (in contrast to lazy-evaluation, explained later). The `ValueContainer` in this example is able to handle constant bit-vectors and all operations involving identity.

Note, that we use only a few operations supported by the `ValueContainer` interface in this example — a digram of the complete interface can be found in Appendix B .

```
1  ValueContainer vc; // initialize new ValueContainer
2  ValueId myVar    = vc.CreateValue(Integer, size: 32b);
3  ValueId eq       = vc.Cmp(x, x, CmpFlags::Equal); // size: 1, value: 1
4  ValueId notEq    = vc.BitNot(eq);                 // size: 1, value: 0
5  ValueId notnotEq = vc.BitNot(notEq);              // size: 1, value: 1
6  tribool test     = vc.IsEq(eq, notnotEq);         // true
```

Algorithm 6: `ValueContainer`: a sequence of immediately evaluated operations

Unsupported Operations and Over-approximation

If a particular implementation of `ValueContainer` does not support some of the operations provided by the `ValueContainer`'s interface, it can simply generate an unknown value — this is a valid although not very useful behaviour leading to an over-approximation. We will now show an example of that behaviour.

In particular, Algorithm 7 shows a bitwise negation of an unknown value, followed by a branch on a condition that the original unknown value is equal to zero. The code fragment utilizes a hypothetical 4 bit-wide integer type `int4_t`.

```
1  int4_t myVar;              // define uninit. var myVar
2  int4_t notMyVar = ~myVar;  // define/assign a bitwise negation of myVar to
       notMyVar
3  if (!myVar) { /*something*/ } // if myVar == 0b0000
```

Algorithm 7: C fragment with bitwise negation and branch on condition

[13] https://youtu.be/_-3Iiads1EM?t=3m40s
[14] http://lists.llvm.org/pipermail/llvm-dev/2016-October/106182.html

The C++ pseudo-code in Algorithm 8 contains a sequence of commands that could be executed to interpret the C code from Algorithm 7.

```
1  ValueContainer vc; // initialize new ValueContainer
2  ValueId myVar = vc.CreateValue(Integer, size: 4b); // value: 0b????
3  ValueId notMyVar = vc.BitNot(myVar);               // value: 0b????
4  // if (!vs.IsTrue(myVar)) -> can we take the branch? we presume yes
5  vc.AssumeFalse(myVar); // myVar === 0b0000, notMyVar ??? 0b1111
```

Algorithm 8: `ValueContainer`: a sequence of commands interpreting code from Alg. 7

Considering again a hypothetical `ValueContainer` capable of working with bit-vectors, a 4 bit-wide unknown integer is created on line 2 of Algorithm 8 and its bitwise negation later on line 3. The command `AssumeFalse` on line 5 can be read as "assume that all the bits are zero" which effectively refines $myVar$ to one possible value at this point.

Now, the status of $notMyVar$ depends on whether the used `ValueContainer` supports operation $BitNot$ (i) only for constant bit-vectors or (ii) for all bit-vectors.

In the former case , $notMyVar$ is an unknown value after both lines 3 and 5, which is a legal result of unsupported operation. From the analyser point of view, it could be any of the 16 possible concrete values representable by a 4 bit-wide integer, leading to an over-approximation.

In the latter case, $notMyVar$ is an unknown value after line 3, but the `ValueContainer` internally maintains its relation with $myVar$. Then, $notMyVar$ is refined to only possible value of 0b1111 on line 5 as a result of $myVar$'s refinement. The price for this precision is a higher complexity of the `ValueContainer` implementation[15].

Lazy vs. Eager Evaluation

In the previous example, $notMyVar$ was either unknown or lazily evaluated based on other values ($myVar$) represented inside the same `ValueContainer`. Third option would be to do an eager evaluation: at every point, generate all possible "concrete" states in order not to loose precision/information.

An example of eager evaluation with the C code from previous example (Algorithm 7) would be splitting the abstract state into 16 new states, where $myVar$ and $notMyVar$ are always a pair of concrete values — it could be done both while interpreting line 1 and line 2. Then, only the state where $myVar == 0b0000$ and $notMyVar == 0b1111$ would follow the execution path taking the branch on line 3.

Although the eager evaluation method essentially eliminates the advantages of abstract interpretation and generally leads to a state explosion, it can be useful in cases where the lazy evaluation is not possible and the loss of precision would be more troublesome for the analysis. Moreover, one can also use only several concrete states (e.g., randomly) turning the analyser into a testing tool.

3.3.4 Using Multiple ValueContainers to Represent a Single Value

Sometimes, the analysis will compute a set of constrains that are not representable in a single supported value domain (implemented as a `ValueContainer`), but could be represented using two separate value domains.

[15] Typical way to implement this would be an tree structure, or a direct utilization of an SMT solver.

An example of such a situation is shown on Algorithm 9: the code evaluates a set of constraints $x \in [10, \infty]$, $y \in [10, \infty]$, $x \neq y$ using an integer interval domain and equality domain (supports equality and non-equality).

```
1   IntervalValueContainer vcInt;
2   EqualityValueContainer vcEq;
3
4   ValueId x = ValueId.GetNext(); // an alternative way to obtain ValueId
5   ValueId y = ValueId.GetNext();
6   vcIntr.CreateValue(Integer, size: 32b, x);
7   vcIntr.CreateValue(Integer, size: 32b, y);
8   vcEq.CreateValue(x);
9   vcEq.CreateValue(y);
10  ValueId const10 = vcIntr.CreateConstantValue(Integer, size: 32b, value:
        10);
11
12  vcIntr.Assume(x, const10, CmpFlags::GreaterOrEqual);
13  vcIntr.Assume(y, const10, CmpFlags::GreaterOrEqual);
14  vcIntr.Assume(x, y, CmpFlags::NotEqual); // no effect!
15  vcEq.Assume(x, const10, CmpFlags::GreaterOrEqual); // no effect!
16  vcEq.Assume(y, const10, CmpFlags::GreaterOrEqual); // no effect!
17  vcEq.Assume(x, y, CmpFlags::NotEqual);
```

Algorithm 9: ValueContainer: multiple containers

The downside of this method is that all queries must be executed against all involved `ValueContainer`s separately.

3.3.5 Composition of ValueContainers

After using multiple `ValueContainer`s separately, the next logical step is to create composite ValueContainer to form a multi-domain representation component. The idea is to provide instruments for semi-automatic construction of composite `ValueContainer`s. Such a composite container works effectively as direct product of its sub-containers as disclosed in more details bellow.

The set of possible concrete values for an abstract value V_{comp} in composite container c_{comp} is an intersection of all possible values V_n across all the sub-containers c_n. All operations are applied to all sub-containers. When querying (`IsCmp`, `IsEq`, etc.) the composite container, the first determinate result from sub-queries is returned.

It is of course also possible to implement a new composite `ValueContainer` manually and optimize based on the specific nature of the used sub-containers — leading to the so called reduced product known in abstract interpretation.

3.3.6 Wrap-up

In the context of ValueContainer, we define abstract values as in following text.

When there is no other information about a value except for its type (e.g. integer, 32 bits wide), we call it an unknown value. The representation of an unknown value can be refined and it can be refined as much as the abstract domain allows — for example, an integer interval domain enables us to refine a value to one possible integer, while doing so for floating-point interval values would be rather complex. Once a representation of an unknown value has been refined, it should no longer be called an unknown value (just an

abstract value). When a value is generated manually from a constant or is a result of an immediate evaluation to a constant, it falls into the constant values category. Constant values and abstract values refined to one possible value belong to the concrete values category (e.g., we consider an integer interval $[5, 5]$ to be a concrete value).

Values in an analysis are identified by `ValueId`. `ValueId` is an identifier that is globally unique. As we can try to represent one value in different abstract domains, we can represent one value in different `ValueContainer`s.

3.4 Analysis Itself

In this section, we start with the relations between different identifiers and mappings of Angie, and continue with description of `StateManager` and other remaining components. Later, we present a new version of analysis loop for abstract interpretation. The algorithm itself has been already presented in the begging of this chapter (see Algorithm 1).

3.4.1 Identifiers linking the Angie IR with Front-end entities

We will now revise, with a help of Figure 3.7, some of the already introduced components in order to make the connections between different Angie components more obvious.

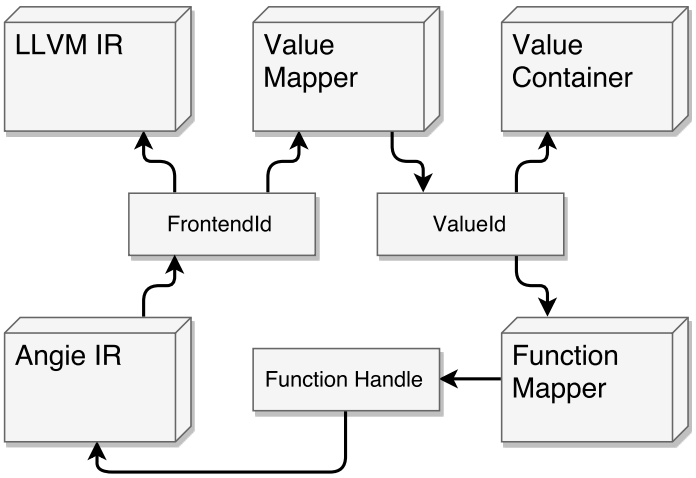

Figure 3.7: Diagram of Angie identifiers and mappings

`FrontendId` is issued to every entity coming from front-end IR that requires addressing, like values and functions.

`FunctionHandle` constits of the entry `CfgNode` of a function, a list of its formal parameters (`FrontendId`s and types), and other implementation-specific info, like function name etc.

`FunctionMapper` is then a bijective mapping between `ValueId`s assigned to functions and `FunctionHandle`s.

The `ValueId`s identifies the values represented inside the `ValueContainer`s.

`ValueMapper` provides a mapping from `FrontendId`s of values to `ValueId`s and also some advanced functionality for dead value analysis and monitoring, which will be described later in this section

3.4.2 StateManager

Whenever a new state is generated during an analysis, many actions has to be carried out (see Algorithm 1). In order to simplify applying those actions, we propose a component named `StateManager` which allows to manage them all in a single place. `StateManager` provides two functionalities:
(i) `TakeNewState` — to take a newly generated state, check if it is not covered by other states and if not, register it with the CFG and add it to the worklist
(ii) `FetchNextState` — to retrieve a next state from the internal managed worklist.

For the `StateManager` to do all the necessary steps from Algorithm 1 for the functionality number (i), it has to be provided with an analysis-supplied entailment-checking, join, widening and abstraction meta-operations. Meta-operation here stand for an operation that is not an abstract counterpart of some concrete instruction and usually is applied to multiple `State`s at once.

The functionality for retrieving next state for processing depends greatly on the actual implementation and settings — it could be a simple DFS (stack) or BFS (queue) solution or a more elaborate one using priorities etc.

Model case of StateManager in action is given in the following example.

> A successor state s_2 for previously existing state s_1 is created. The program counter of s_2 points to a location s_2.loc and there is already a state s_3 registered with the location s_2.loc.
>
> Upon the request to process new state, the `StateManager` performs the entailment-check on s_2 with regards to s_3. The result is that s_2 is not covered by s_3, so s_2 is registered with the location s_2.loc and inserted into the worklist.
>
> The `StateManager` implementation in question uses stack (DFS) for the underlying worklist — when the analysis asks for a new state to process, s_2 is returned as the last one inserted.

The interface of `StateManager` is in the left part of Figure 3.8.

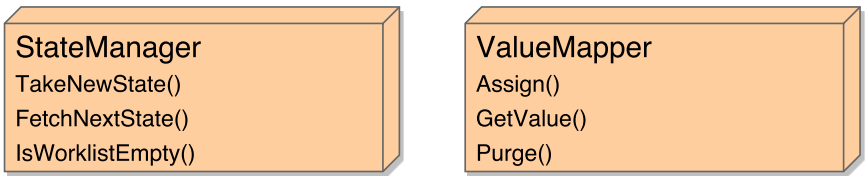

Figure 3.8: An interface of `StateManager` (left) and `ValueMapper` (right)

3.4.3 ValueMapper in Details and the Liveness Analysis

Because of the scope-less nature of intermediate representation used in Angie[16], the analysis does not get any "free" liveness information about the SSA registers. That is a critical problem for loops — while the variables in the original scoped code could be at least automatically "killed" at the end of each loop iteration, it is not possible in Angie IR without previously performing liveness analysis on the CFG. Fortunately, the liveness analysis for code in SSA form only requires two passes trough the CFG [3].

`ValueMapper`, in fact, represents a configuration of SSA registers of the analysis state, which makes it the best component to monitor and control the liveness of values (SSA registers).

We propose an in interface — shown in the right part of Figure 3.8 — with methods

- `Assign` — registers ValueId with the FrontendId,
- `GetValue` — reads the associated ValueId,
- and `Purge` — unregistered any SSA registers (FrontendId) not alive at the current program location and triggers implantation-defined actions in other sub-components of `State`.

[16] Angie IR is based on LLVM IR — both are in partial SSA form, see Section 2.4

The method `Purge` is excepted to be used before an abstraction is attempted. Also, Purge will be triggered internally whenever a `FrontendId` would have been re-assigned (e.g. in loops). Note that this feature has not been yet implemented in any form and its design might need to be changed.

3.4.4 State

`State` is an implementation of abstract program configuration (state) in Angie. It connects together a set of information consisting of program location, its `State` predecessors, the configuration of SSA registers (identified by `FrontendId`s), the current state of used `ValueContainer`s, call stack and any analysis-specific representation of some aspect of the interpreted program (like memory, file descriptors, etc.).

The enumeration of primary properties of a State interface are shown on Figure **??**, while the description is given in following text.

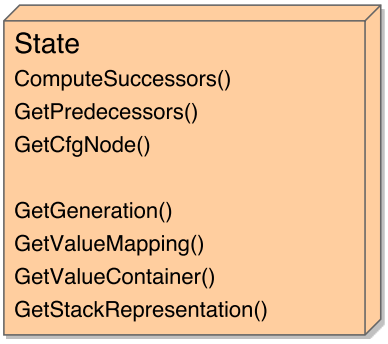

Figure 3.9: `State` interface

The *ComputeSucessors* method allows for implementation-defined process of obtaining successors `State`s, making the interface of State generic enough to work with other loops beside abstract-interpretation-based.

The implementation of *ComputeSucessors* for abstract interpretation would be following:

> Retrieve the associated `CfgNode` c for the state s with s.GetCfgNode(). Then retrieve the `Operation` o to be performed via c.GetOperation() together with the necessary arguments A via c.GetArguments().
>
> Execute o with s and A. The result is a set of newly computed `State`s S_{succ} which is returned to the caller of *ComputeSucessors*.

`GetPredecessors` returns the list of `States` this state originated from (in case of joinformed `State`, there might a multiple of them)

`GetCfgNode` returns an equivalent of program counter, the next `CfgNode` to be interpreted

`Clone` will make a deep clone of the state which can be modified. Cloning and modyfing the state is the preferred way to create successors.

`GetGeneration` is for determining the generation number of the state (initial state has 0 and the number grows by 1 with every successor generation).

`GetValueMapping` retrieves the current `ValueMapper`.

`GetCallStack` gives access to the current abstract call stack.

Finally, an example showing most of the described methods in use is given in Algorithm 10. Since an arithmetic operation on the level of Angie IR does not have any side-effects, we can implement it without any analysis-specific tools, just using `ValueMapper` and `ValueContainer`.

```
1    CfgNode toExecute    = oldState.GetCfgNode()
2    Operation op         = toExecute.GetOperation();
3    OperArg args         = toExecute.GetArguments();
4
5    op.Execute(oldState, args); // op is an OperationBinaryArtihmeticOp
6    ... // begin: implementation of OperationBinaryArtihmeticOp.Execute
7    State newState       = oldState.Clone();
8    // here, some code setting new program counter in newState, etc.
9
10   ValueMapper mapper   = newState.GetValueMapper();
11   ValueContainer vc    = newState.GetInnerValueContainer();
12
13   BinaryOpOptions opts = args.GetOptions();
14   Type type            = args.GetOperand(0).type;
15   ValueId lhs          = mapper.GetValue(args.GetOperand(0).id);
16   ValueId rhs          = mapper.GetValue(args.GetOperand(1).id);
17   ValueId retVal       = vc.BinOp(lhs, rhs, type, opts);
18
19   mapper.Assign(args.GetTarget().id, retVal);
20   return Vector<State*>}{newState};
21   ... // end: implementation of OperationBinaryArtihmeticOp.Execute
```

Algorithm 10: Complete code of an `Operation`

3.4.5 Angie Analysis Loop

An analysis loop from Algorithm 1 decomposed into the presented Angie components is shown in Algorithm 11.

Input: Angie IR CFG
Output: labelled CFG where each `CfgNode` is labelled by a set of `State`s governed by a `StateManager`
Data: `StateManager` governed worklist of `State`s containg only the initial state s_0
Result: abstract state space of the analysed program has ben explored

```
1 while not StateManager.IsWorklistEmpty() do
2 |   State s_old ← StateManager.GetState();
3 |   list of States S_new ← s_old.ComputeSuccessors();
4 |   foreach s_new ∈ S_new do
5 |   |   StateManager.TakeNewState(s_new);
6 |   end
7 end
```

Algorithm 11: Angie analysis loop

Note that the *ProcessNewState* could also be reworked so it consumes a list of states instead of one in a time, potential improving efficiency for some meta-operations.

The shown design of control loop enables different kinds of analyses, as long as they provide implementations of `StateManager` and `State`'s *ComputeSucessors*.

3.4.6 Inter-procedural Analysis

During a standard native execution of programs, function calls and returns are translated into a series of stack-memory and registers writes and reads followed by a jump. The machine registers are copied onto the stack either by the caller or the callee, and the rest of program variables is implicitly implemented on a stack frame of the active function.

The described techniques provide the running application with:

- separation of variable sets during recursion (direct or indirect),
- a place for return value,
- a place for return jump address.

The so-called variable set corresponds to Angie's mapping of `FrontendId`s (SSA registers and constants) to abstract values implemented by the `ValueMapper` and the program address would translate to a `CfgNode`.

There are two components supporting procedural programs in Angie. The first is a `StackFrame`, consisting of `ValueMapper`, `FrontendId` of function-call result value and `CfgNode` of return location. The second is then a `CallStack`, a list of `StackFrame`s with an interface to access the top-most stack frame.

Chapter 4

Implementation and Experiments

The prototype implementation of Angie is currently hosted on GitHub Git repository and is targeting C++ 14 and is tested with a minimum of GCC 4.9 / MSVC 2015.

The repository has working continuous integration hooks, stable master branch, unstable feature and develop branches, and contains all the information (dependencies, install instructions) necessary to get the prototype working. The source code is provided under the GNU LGPL licence v3+.

The code is written with regards to C++ Core Guidelines and arranged into many source and header files. Unfortunately, due to an excessive use of templates in the analysis code, most of the analysis-connected code is being compiled into single module. Project management and compilation is implemented via CMake.[1]

Most currently used third party libraries (exception is, for example, Google Test) are header-only. One of those, Range-v3[2], is worth mentioning - it is based on the current version of Ranges and Concepts drafts for Technical Specifications[3]. It allows non-owning selection, transformation and also generator views to be constructed as single objects using functional-like syntax, enabling easy-to read constructs and re-use of algorithms similar to <functional>[4].

4.1 Common Basic Components

Amongst the common components used through-out Angie, which can be found in files *{General, Definition, Exceptions, Type, IdImpl, ValueId}.hpp*, belong `FrontendId`, `ValueId`, all kinds of exceptions, `OperArg`, `OperationArgs` and `Type`. Various flags and kinds enums, using aliases, and macros are also defined in the mentioned files, particularly in *General.hpp* Various utilities for file operations etc. are placed in *OsUtils* module.

`FrontendId`, `ValueId` are adhering to the presented design and are implemented — together with other Ids used in Angie — using `IdImpl` template.

`OperArg` and `OperationArgs` implement a mechanism for passing generic arguments to `Operations`. `OperationArgs` class can be legally converted to one of more specific types, based on the executed `Operation`s.

[1] A semi-informal set of guidelines for writing C++ maintained by Bjarne Stroustrup & Herb Sutter, published at https://github.com/isocpp/CppCoreGuidelines

[2] https://github.com/ericniebler/range-v3

[3] TS are papers for features that are proposed to be merged in one of the next ISO C++ standards.

[4] A part of C++ standard library containing standard algorithm functions working with iterators

As for `Type`, Angie contains two implementations of types. The LLVM-based is designed to be used during standard operation of Angie — it translates the request between the analysis and the front-end without creating any intermediate type systems. The second one is implemented without any external dependencies and is designed for experiments and testing of abstract domains — an activity which generally does not need working LLVM. The implementations can be switched using the `TYPE_KIND` define.

4.2 ValueContainers

`IValueContainer` (the interface for `ValueContainers`) is implemented in file Values.hpp. Many methods of the interface are implemented using their more generic counterparts contained in the same interface and many other methods are implemented using NEI (not implemented exception) — that allows the developers of `ValueContainer`s to implement only the necessary minimum first, test it with simple analysis and then finish the implementation, potentially optimizing the work of container.

`ValueContainerV1` is an experimental implementation[5] of integer interval combined with simple binary relational abstraction domain.

ValuesZ3.hpp contains an incomplete proof-of-concept implementation of Z3[6] SMT solver adapter for ValueContainer interface, using the bit-vector theory as backing for integer representation. The C++ interface is based on using a single Z3 context for storing all Z3 objects (variables, expressions) and creating a new solver for every query. The container creates the representations of values directly in a Z3 context, while it also creates a local database of constraints created. When a query is to be processed by the container, it first uses to local database to fill a newly created solvers with appropriate existing expressions (constraints), then it inserts the condition to be tested and checks the satisfiability of the expression set.

4.3 Front-end Adapter and Related Components

`CfgNode`, `Operation`, `OperationFactory`, `ValueMapper` and `FunctionMapper` components — which are used by both front-end adapter and analysis loop — are implemented in files *{ICfgNode, IOperation, FrontendValueMapper}.hpp* and related *.cpp* files.

The interface of `CfgNode` component is implemented in the `ICfgNode` abstract class, with one exception: instead of `GetOperation`, `ICfgNode` provides an `Execute` method which executes the `Operation` with `OperationArgs` stored inside the node.

The abstract class `CfgNodes` provides a basis for implementation of `CfgNode` as pointer-based, separately allocated nodes. The full implementations of `ICfgNode` in the Angie prototype are front-end specific due to the need to access debugging/source code information.

The basic functionality of `ValueMapper` is implemented in `Mapper` class. The value life control functionality is not implemented in the prototype as the necessary liveness analysis, which is a prerequisite, is not implemented either.

[5] `VCv1` has been implemented by Michal Charvát under the supervision of the author & the supervisor of thesis. His contributions are clearly marked in the source code and can be tracked in the source code repository.

[6] https://github.com/Z3Prover/z3

The algorithms parsing the LLVM IR into Angie IR are implemented in files *{LlvmFrontend, LlvmGlobals}.{hpp, cpp}*. The individual parsers / parsing blocks (visualised in Figure 3.6) are implemented in functions with corresponding name.

4.4 Analysis Loop, State and StateManager

`State`, `StateManager` and the analysis loop implemented in the prototype slightly differ from their models described in the design chapter. Moreover, to minimize amount of boilerplate code in implementations of `Operation` shown in Algorithm 10, the Angie prototype provides various templates in file *Operations.hpp*.

The most notable difference, except for different method names, is that the `ComputeSuccessors` method (Figure 3.9) is not a part of the `IState` interface in the prototype implementation — instead, its functionality is implemented partly in the analysis loop (retrieving the `CfgNode`) and partly in mentioned templates for `Operations`. Also, `CreateSuccessor` replaces the `Clone` method in the prototype and the `CallStack` is not implemented as a part of the framework.

Further, the `State` component is implemented in three layers, (i) an interface `IState`, (ii) abstract class State providing basic implementation and (iii) analysis-specific class.

4.5 Proof-of-concept Analysis

The forward null analysis (*ForwardNullAnalysis.hpp*) is a proof-of-concept analysis: it does not have a complete support for inter-procedural analysis and comes only with a limited representation of stack. It is mainly capable of identifying null dereferences in a scope of one function.

However, thanks to its simplicity, it demonstrates how to implement a new analysis in Angie: one must create an analysis-specific implementation of `State` together, approximately 8 `Operations` and an `OperationFactory`. The rest of `Operations` (mainly those without side-effects) are implemented in a generic way in *Operations.hpp*.

4.6 SMG-based Analysis

The main experimental analysis in the Angie prototype is implemented in file *MemoryGraphAnalysis.hpp* and various other files prefixed with *Smg*. Amongst the important features of this analysis belongs symbolic memory graph representation of memory, full support for intra-procedural analysis and `Operations` implementing *malloc* and *free* C Standard Library functions.

The analysis is able to detected null pointer dereference, invalid pointer dereference, invalid free and supports a subset of Predator intrinsic functions and comes with an experimental SMG plotting support.

However, the implementation of symbolic memory graphs is limited to regions only and has only partial support for data reinterpretation (partially overlapping values do not affect each other), which might lead to incorrect analysis. Nevertheless, the emphasis was put on the development of essential graph structures and algorithms, to allow for easy implementation of maintainable abstraction algorithms.

4.6.1 Modifications to the Structure of SMGs

The structures implementing the SMGs differs in some basic aspects from the one presented in [7].

Firstly, there are generally two options of implementing the connections between SMG entities in C++: either storage containers like vector together with the use of identifiers, or separate allocation of entities with the use of pointers — we opted for identifiers, because they do work well with copy-on-write and are more suitable for deep-cloning of graphs.

One of the major differences is the elimination of value nodes as separate entities. Since the ValueContainer is a preferred way of representation of scalar values in Angie, the SMG values can be simplified to only a `ValueId`. Moreover, the target specifier of points-to-edges is replaced by a special wrapper nodes, or ports, that identify the particular kind of access to the target Object. These modifications enabled us to merge the original separate has-value edge, value node and points-to edge into a single tuple of source object ID, source offset, `ValueId` of the value, target object ID and target offset, where the type of value is for pointer-values is just a generic pointer.

4.7 Usage

The prototype itself accepts an input in the form of LLVM IR files (either binary or text) and prints all detected errors on standard output. Also, printing of the interpreted LLVM IR instructions can be switched on for debugging purposes. The LLVM IR files can be obtained from C source files using the accompanying script.

For more enhanced debugging experience, it is possible to insert special trap into the input LLVM IR which triggers a software breakpoint in the tool upon interpretation. The trap has a form of a special metadata node `angie.debugbreak` and its use is shown in Figure 4.1

```
%7 = alloca %struct.main_struct_s, align 4
%8 = alloca %struct.main_struct_s, align 4, !angie.debugbreak !{}
```

Figure 4.1: Example of debugging trap placed in LLVM IR text file

4.8 Experiments

Apart from the experiments with the original Predator tool, which we performed to get acquainted with the tool and to optimize it for the SV-COMP. While the former experiments were presented in Chapter 2, we are now going to describe our experiments with the Angie platform.

The implementation comes with two sets of small examples which focus on various features of the framework and the analyses. Further, the prototype was also tested on a part of Predator test-suite.

4.8.1 Basic examples

The first set `01_mincase` consists of 6 small examples of 2 errors (dereferencing null pointer and dereferencing uninitialized pointer) and also a potentially dangerous use of an unini-

tialized value. Examples 1 to 4 do not emit any branches in the LLVM IR, which makes them very useful for an initial analysis experiments. Examples 5 and 6 then add branching and function calls, respectively. A filtered output of example 5 analysis is shown in Figure 4.2

```
$ ../tmp/angie.exe -f 01_mincase_05_all_conditional.ll
ex2/01_mincase_05_all_conditional.c:22:12
Error: Program tried to dereference a null pointer.
ex2/01_mincase_05_all_conditional.c:27:12
Error: Program tried to dereference an invalid location.
```

Figure 4.2: Filtered output of prototype implementation running SMG-based analysis

4.8.2 Heap examples

The second set `02_heap` comprises of 4 examples focused on heap allocation and pointer manipulation, targeting the experimental SMG-based analysis. While the first 3 examples contain loops with a limited number of iterations, the number of iterations is an unknown value in the example 4 and the tool never terminates because of the missing abstraction. Also note that example number 3 (doubly linked list constructions) comes with a variant plotting the internal graph representation during the analysis.

4.8.3 Predator test-suite

The prototype was also put through a testing with a set of 224 tests from Predator test-suite (namely from directory `tests/predator-regre` in Predator repository) and a time-limit of 5s (discovered experimentally).

Out of all the examples, the tool did not terminate in 103 cases, exited on an internal exception in 83 cases, did not detect any errors in 27 cases, and detected some error in 17 cases.

Some of the 103 tests which did not terminate appears to be finite but did not terminate even with increased time (test-0014, 0016, ...), but most contain in infinite loop (0001, 0004, 00013, 0047, ...).

Most of the 83 cases of internal exception were caused by failure to compile the test (total 20), missing external (stdlib) function (total 21) or unsupported instruction phi or memcpy (total 24). Other cases include incomplete support for global variables, unsupported constant type or cast instruction. Note that the phi instructions (generated usually from C ternary operator) could be replaced with the transformation passes described in [15].

The cases with errors detected included a few false alarms, due to imprecise ValueContainer operations (non-equality edges, 0025) or due to missing aliasing/data-reinterpretation support (0075, 0076).

Finally, out of the 27 test that supposedly do not contain any error, the analyser missed only two errors of out-of-bounds dereference (0009, 0077).

Considering the current state of the implementation, we find the amount of correctly analysed test-cases a success.

Chapter 5

Conclusion

Nowadays the world has a high demand for a software whose correctness and proper operation is a question of life and death. This being a fact, a program verification methods have become a very important area of research.

To improve the conditions for developers of new program analysers, we are working on a framework for static analyses called Angie. We have presented a complete design of the framework in an incremental way, followed by a description of the current state of the implementation.

As for the future development, we hope to get the implementation in compliance with the current design proposal, improve the existing implementations of value domain representations and to extend the support of LLVM intrinsic functions and C Standard Library functions. Also, creating a working liveness analysis and other progress towards working abstraction in *symbolic memory graphs* would be a reasonable courses of action.

Bibliography

[1] *LLVM Language Reference*. Avialable at: http://llvm.org/docs/LangRef.html [retrived on 2017-04-10].

[2] Beyer., D.: *Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)*. Proc. TACAS, Springer, 2016.

[3] Boissinot, B.: *Towards an SSA based compiler back-end: some interesting properties of SSA and its extensions*. PhD. Thesis. Lyon, École normale supérieure. 2010.

[4] Cousot, P.: Abstract interpretation based formal methods and future challenges. In *Informatics*. Springer. 2001. pp. 138–156.

[5] Cousot, P.; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977. pp. 238–252.

[6] Cousot, P.; Cousot, R.: Abstract interpretation frameworks. *Journal of logic and computation*. vol. 2, no. 4. 1992: pp. 511–547.

[7] Dudka, K.; Peringer, P.; Vojnar, T.: *Byte-Precise Verification of Low-Level List Manipulation*. In *Proc. of SAS'13*, *LNCS* 7935, pages 214–237, Springer, 2013. An extended version avaliable at: http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/FIT-TR-2012-04.pdf.

[8] Dudka, K.; Peringer, P.; Vojnar, T.: An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proceedings of the 13th International Conference on Computer Aided Systems Theory*. The Universidad de Las Palmas de Gran Canaria. 2011. ISBN 978-84-693-9560-8. pp. 328–329.

[9] Kotoun, M.; Peringer, P.; Šoková, V.; et al.: *Optimized PredatorHP and the SV-COMP Heap and Memory Safety Benchmark (Competition Contribution)*. In *Proc. of TACAS'16 as a competition contribution within SV-COMP'16*, *LNCS* 9636, pages 942–945, Springer, 2016. An extended version is available at: http://www.fit.vutbr.cz/~vojnar/Publications/predators-svcomp-16.pdf.

[10] Kotoun, M.; Peringer, P.; Šoková, V.; et al.: *PredatorHP and SV-COMP 2017 (Competition Contribution)*. http://www.fit.vutbr.cz/research/groups/verifit/tools/predator-hp/download/predator-svcomp-2017.pdf.

[11] Křena, B.; Vojnar, T.: Automated formal analysis and verification: an overview. *International Journal of General Systems*. vol. 42, no. 4. 2013: pp. 335–365.

[12] Muller, P.; Peringer, P.; Vojnar, T.: *Predator Hunting Party (Competition Contribution)*. In *Proc. of TACAS'15 as a competition contribution within SV-COMP'15*, *LNCS* 9035, pages 443–446, Springer, 2015.

[13] Sampson, A.: *LLVM for Grad Students*. August 2015. avialable at: https://www.cs.cornell.edu/~asampson/blog/llvm.html [retrived on 2017-04-10].

[14] Šoková, V.: *Vývoj LLVM adaptéru pro infrastrukturu Code Listener*. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. 2013.

[15] Šoková, V.: *Analýza práce s dynamickými datovými strukturami v C programech*. Master's Thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. 2015.

Appendices

Appendix A

Storage medium

The accompanying medium contains electronic version of this report and source code files of Angie including a version control repository.

Appendix B

ValueContainer Interface